並列論理型言語処理系KLICによるPARIの並列化(数式処理における理論とその応用の研究)

AUTHOR(S):

藤瀬, 哲朗

CITATION:

藤瀬, 哲朗. 並列論理型言語処理系KLICによるPARIの並列化(数式処理における理論とその応用の研究). 数理解析研究所講究録 1995, 920: 149-160

ISSUE DATE:

1995-08

URL:

http://hdl.handle.net/2433/59708

数理解析研究所講究録
920 巻 1995 年 149-160

16.

並列論理型言語処理系 KLIC による PARI の並列化

藤瀬 哲朗 (ICOT)

16.1 はじめに

代数計算を並列化する試みは数多くみられるが、並列数式処理系に関する話題はほとんどみられない。行列の計算を分解して集積するような単純分散による計算のために、複数の数式処理系を単純に接続して計算実験に利用することはそれほど珍しいことではない。しかし探索問題のような複雑な処理や負荷分散制御等が必要な計算を実現するためにははにはきちんとした並列処理環境を備えた数式処理系が必要になる。

数式処理系 PARI [Cohen 93] はライブラリベースの高速な数式処理系である。数式の操作に関する機能はそれほど豊富ではないが、数多くのアルゴリズムを記述した計算パッケージをもつ。PARI は一般のプログラムとリンクすることで数式計算のサービスを提供することができる。今回この PARI を利用した並列代数計算環境を、並列論理型言語 KL1[Ueda & Chikayama 90] のポータブル実装である KLIC [Fujise 94] とリンクすることで実現した。これによって PARI のもつ高速な計算機能と KL1 言語のもつ並列記号計算の記述性を活用した並列代数計算環境を構築した。

具体的には KLIC のもつ KL1 言語の意味論を壊さずに他言語プログラムと取り込むための枠組 (ジェネリックオブジェクトと呼ばれる) を利用し、数式処理系 PARI を KL1 言語の枠組に収めた。

本稿では代数計算を単純に並列化できない例として、[Murao & Fujise 94] で述べられている非決定的計算を伴う数式計算を挙げる。そのような非決定的計算が並列論理型言語 KL1 により容易に記述できることを示す。そして KLIC とジェネリックオブジェクトについて簡単に触れた後、PARI の並列化とそれによる計算例を示す。

16.2 非決定的計算と数式処理

非決定的計算の典型的例題は探索問題である。代数的な意味での枝刈りを必要とする探索問題の例として Feedback Connection Polynomial の Distinct Order Factorization を挙げる。

p : 素数 $s.t.\ p > 2$, $\Lambda(z) \in Z_{[}z]$, ξ : 1 の原始 $(p-1)$ 乗根とする。その時 Feedback Connection Polynomial $\Lambda(z)$ は、

$$\begin{aligned} \Lambda(z) &= \prod_{i=1}^{t}(z-\alpha_i) \ \ mod\ \ p \\ &= \prod_{i=1}^{t}(z-\xi^{k_i}) \ \ mod\ \ p. \end{aligned}$$

なる性質をもつ。この多項式について次の問題を考える。

問題 1. [Distinct Order Factorization]

p : 素数 $s.t. p > 2, G : (p-1)$ の因子, $\xi \in Z_{\tau}$ かつ $\Phi(z) \in Z_{[}z] s.t. \Phi(z)$ は線形因子の積であることがわかっていることとする。そのとき次の集合 L を求める。

$$L = \{(k, \phi_k(z)) | \phi_k = gcd(\Phi(z), z^{(p-1)/G} - \xi^{k(p-1)/G})\}$$

$\phi(z)$ を $\Phi(z)$ に、G を p の他の因子で置き換え、再帰的にこの計算を行なうことにより、すべての線形因子を求めることができる。この計算は Black Box 多項式の同定問題の一部であり、詳細は文献 [Murao & Fujise 94] に譲る。さて Distinct Order Factorization の逐次アルゴリズムを以下に示す。

アルゴリズム 16.2.1. (Distinct Order Factorization (Sequential Version))

$Q \Leftarrow (p-1)/G$; $\quad A \Leftarrow \xi^Q \bmod p$; $\quad B \Leftarrow z^Q \bmod \Phi(z)$;
if $\deg \Phi(z) = 1$, *e.g.*, $\Phi(z) = z - \alpha$ **then return** $\{[\log_A(\alpha^Q), \Phi(z)]\}$;
$L \Leftarrow \{\}$; $\quad k \Leftarrow 0$; $\quad C \Leftarrow \Phi(z)$;
while $k < G - 1$ and $C \neq 1$ **do**
  if $\deg B = 0$ **then return**$\{[\log_\xi B, C]\} \cup L$;
  $w \Leftarrow \gcd(C, B - A^k) \bmod p$;
  if $w \neq 1$ **then**
    $L \Leftarrow \{[k, w]\} \cup L$; $\quad$ % add a pair $[k, w]$ to L.
    $C \Leftarrow C/w \bmod p$;
    if $\deg C = 1$, *e.g.*, $C = z - \alpha$ **then return** $\{[\log_A(\alpha^Q), C]\} \cup L$;

if $C \neq 1$ **then** $B \Leftarrow B \bmod C$;

$k \Leftarrow k+1$;

return L;

このアルゴリズムは確率的である。なぜなら k が小さいうちに $degB = 0$ もしくは $C = 1$ となった場合、すなわち全因子が求まった時点で計算が終了する。因子が $k = G-1$ でみつかった場合は、この **while** ループはほとんどすべての k に対する処理を行なうこととなる。

このアルゴリズムを単純に並列化する。並列化の指針として単純に **while** ループ を分割して複数個の worker に仕事を分けると次のようにできる。

アルゴリズム **16.2.2. (Distinct Order Factorization (Parallel Version))**

$Q \Leftarrow (p-1)/G$; $\quad A \Leftarrow \xi^Q \bmod p$;

if $\deg\Phi(z) = 1$, *e.g.*, $\Phi(z) = z - \alpha$ **then return** $\{[\log_A(\alpha^Q), \Phi(z)]\}$;

$S \Leftarrow \lceil \frac{G-2}{N} \rceil$;

for i := 0 **to** $N-1$ **do_parallel**

$k_i \Leftarrow S \cdot i$; $\quad G_i \Leftarrow min(G-1, k_i + S)$;

$L_i \Leftarrow \{\}$; $\quad B_i \Leftarrow z^Q \bmod \Phi(z)$; $\quad C_i \Leftarrow \Phi(z)$;

while $k_i < G_i - 1$ and $C_i \neq 1$ **do**

if $\deg B_i = 0$ **then**

$L_i \Leftarrow \{[\log_\xi B_i, C_i]\} \cup L_i$; $\quad$ **exit_do_parallel**;

$w_i \Leftarrow \gcd(C_i, B_i - A^{k_i}) \bmod p$;

if $w_i \neq 1$ **then**

$L_i \Leftarrow \{[k_i, w_i]\} \cup L_i$; $\quad$ % add a pair $[k_i, w_i]$ to L_i.

$C_i \Leftarrow C_i / w_i \bmod p$;

if $\deg C_i = 1$, *e.g.*, $C_i = z - \alpha$ **then**

$L_i \Leftarrow \{[\log_A(\alpha^Q), C_i]\} \cup L_i$; $\quad$ **exit_do_parallel**;

if $C_i \neq 1$ **then** $B_i \Leftarrow B_i \bmod C_i$;

$k_i \Leftarrow k_i + 1$;

return $\bigcup_{i=0}^{N-1} L_i$;

この並列化したアルゴリズムによる計算は、小さい k によりすべての解が求まる場合に、次の理由で逐次アルゴリズムによるものよりひどく遅くなる可能性がある。

・ 解がみつかる度に B および C の次数が下がることとなり、逐次アルゴリズムでは、早いうちにループ脱出条件を充たすために計算が短時間で終了する。それに対して並列アルゴリズムでは解

の計算が複数箇所に散るため、ループ脱出条件を検出できない。

- 同様に本アルゴリズムにおいて最も計算時間に影響する GCD 計算時間が、逐次アルゴリズムでは C の次数減少により解がみつかる度に短くなる。それに対して並列アルゴリズムでは、次数減少による恩恵は解がみつかったループ内でしか被らない。

表 1 に [Murao & Fujise 94] に挙げられている台数効果が著しく劣化する例を載せる。

表 1　100 次の多項式の因数分解 (1)

worker 数	1	2	4	8	12	16
台数効果	1.0	0.0	0.0	0.1	0.2	0.3

このアルゴリズムを改善する。まず終了条件による改善である。

計算の終了条件は全解の次数の和が元の多項式 $\Phi(z)$ の次数と等しくなることである。各ループで得られる解の集合ではなく各解が求まるたびにループの実行終了に関わらずそれらを集積し、求まった解の次数の合計が元の多項式の次数に一致した時に終了フラグを挙げる。この終了フラグをループ毎で検査することで処理終了を理解する。終了フラグが挙がった場合は、各ループでの計算不要であることを意味するので各ループが終了する。以下にアルゴリズムを示す[1]。

アルゴリズム 16.2.3. (Distinct Order Factorization (Revised Parallel Version (1)))

$Q \Leftarrow (p-1)/G$;　　$A \Leftarrow \xi^Q \bmod p$;
if $\deg \Phi(z) = 1$, *e.g.*, $\Phi(z) = z - \alpha$ **then return** $\{[\log_A(\alpha^Q), \Phi(z)]\}$;
$S \Leftarrow \lceil \frac{G-2}{N} \rceil$;　　$L \Leftarrow \{\}$;
$fin\_flag \Leftarrow off$;
for i := 0 **to** $N-1$ **do_parallel**
　$k_i \Leftarrow Si$;　　$G_i \Leftarrow min(G-1, k_i + S)$;
　$B_i \Leftarrow z^Q \bmod \Phi(z)$;　　$C_i \Leftarrow \Phi(z)$;
　while $k_i < G_i - 1$ and $C_i \neq 1$ **do**
　　if $fin\_flag = on$ **then** exit_do_parallel;
　　if $\deg B_i = 0$ **then**
　　$L \Leftarrow \{[\log_\xi B_i, C_i]\} \cup L$;　　**exit_do_parallel**;
　　$w_i \Leftarrow \gcd(C_i, B_i - A^{k_i}) \bmod p$;
　　if $w_i \neq 1$ **then**
　　$L \Leftarrow \{[k_i, w_i]\} \cup L$;
　　$C_i \Leftarrow C_i / w_i \bmod p$;

[1] ここでは良い記述法がみつからなかったので、並列部分を **for** A **do_parallel** B **and** C ··· という構文により A という範囲の下で B および C から **and** で連なる処理を並列に実行することを意味するようにした。

if $\deg C_i = 1$, *e.g.*, $C_i = z - \alpha$ **then**

$L \Leftarrow \{ [\log_A(\alpha^Q), C_i] \} \cup L$; **exit_do_parallel**;

if $C_i \neq 1$ **then** $B_i \Leftarrow B_i \bmod C_i$;

$k_i \Leftarrow k_i + 1$;

and while *true* **do**

if $(\deg \Phi(z) = \sum_{c_i \in \{e \mid (K,e) \in L\}} \deg c_i)$ **then** $fin\_flag \Leftarrow on$; **return** L;

このアルゴリズムは、前述の並列化への弊害のうち C の次数の減少による GCD 計算時間の短縮化には寄与しない。そこでループで求められつつある因子を他ループに broadcast し、因子を受信したループは適当な段階でその因子により C および B を除すことにより、GCD 計算時間を短くできる。また各ループも終了条件に早期に到達し易くなる。改良されたアルゴリズムはつぎの通り。

アルゴリズム **16.2.4. (Distinct Order Factorization (Revised Parallel Version (2)))**

$Q \Leftarrow (p-1)/G$; $A \Leftarrow \xi^Q \bmod p$;

if $\deg \Phi(z) = 1$, *e.g.*, $\Phi(z) = z - \alpha$ **then return** $\{ [\log_A(\alpha^Q), \Phi(z)] \}$;

$S \Leftarrow \lceil \frac{G-2}{N} \rceil$; $L \Leftarrow \{\}$;

$fin\_flag \Leftarrow off$;

for i := 0 **to** $N-1$ **do_parallel**

$k_i \Leftarrow Si$; $G_i \Leftarrow min(G-1, k_i + S)$;

$B_i \Leftarrow z^Q \bmod \Phi(z)$; $C_i \Leftarrow \Phi(z)$;

while $k_i < G_i - 1$ and $C_i \neq 1$ **do**

if $fin\_flag = on$ **then** exit_do_parallel;

if 新しい $w_j (j \neq i)$ がみつかった **then**

$C_i \Leftarrow C_i / w_j \bmod p$; $B_i \Leftarrow B_i \bmod C_i$;

if $\deg B_i = 0$ **then**

$L \Leftarrow \{ [\log_\xi B_i, C_i] \} \cup L$; **exit_do_parallel**;

$w_i \Leftarrow \gcd(C_i, B_i - A^{k_i}) \bmod p$;

if $w_i \neq 1$ **then**

$L \Leftarrow \{ [k_i, w_i] \} \cup L$;

$C_i \Leftarrow C_i / w_i \bmod p$;

if $C_i \neq 1$ **then** $B_i \Leftarrow B_i \bmod C_i$;

$k_i \Leftarrow k_i + 1$;

and while true **do**

if $(\deg \Phi(z) = \sum_{c_i \in \{e \mid (K,e) \in L\}} \deg c_i)$ **then** $fin\_flag \Leftarrow on$; **return** L;

この処理は探索問題の各探索計算のコストを削減するという意味でここでは**代数的枝刈り**と呼ぶ。代数的枝刈りに関して次の性質に注意すべきである。

・ この代数的枝刈りは計算における効率化に寄与するものである。別に改善に関する処理が為されなくとも数学的には正しい解を求めることができる。

代数的枝刈りは行なわれても行なわれなくとも代数的には問題がないのである。

代数的枝刈りのためにループ間や解の回収処理と各ループの間で因子の通信が発生する。しかし前述の性質によりこの通信は行なわれないもしくは因子が遅れて到着 / 処理されても問題がない。つまり代数的には決定的であるが、計算としては非決定性を持ったアルゴリズムであると言える。特にこの計算では到着の遅れを許すものとなっている。

このような代数的正当性が証明できている非決定性アルゴリズムは、現在の共有メモリおよび分散メモリ並列マシンにおける通信の遅れを許容するアルゴリズムであると言える。物理プロセッサ性能が高い場合は少い通信で処理可能であり、通信性能が高い場合は worker 数を増やすことで計算時間を短くできる可能性があるわけである。

以上のように並列代数計算アルゴリズムに非決定性をもたせることにより、並列にかつ多くのアーキテクチャに対応できる可能性をもつ計算が実現できるわけである。この非決定的アルゴリズムが記述できるのが特徴であると言われているのが並列論理型プログラミング言語である。

16.3 並列論理型言語 KL1 と非決定的計算

■並列論理型言語 KL1

KL1 は並列論理型言語のひとつである。KL1 は証明系としてみると完全ではないが[2]、その代償として非決定的処理が記述できるという非常に強力な言語機能を持ち合わせている。

KL1 は細粒度並列記号処理向けプログラミング言語といった方がわかり易いであろう。KL1 のプログラムは図 1 に示すようなガード付き Horn 節の集合である。

$$\underbrace{Head :- Guard_1, Guard_2, \cdots, Guard_m}_{\text{ガード部}} \mid \underbrace{Body_1, Body_2, \cdots, Body_n}_{\text{ボディ部}}$$

図 1 KL1 節

[2]もちろん逐次 Prolog も完全ではない。

| (コミット演算子) の前部をガード部、後部をボディ部と呼ぶ。ガード部はボディ部を実行するための適用条件を表している。*Head*、*Guard* および *Body* はそれぞれ述語であり、図 1 の節は述語 *Head* の定義のひとつである (この節は候補節とも呼ばれ *Head* について同じ述語名、同じ引数の候補節群が述語定義となる)。KL1 では述語の計算は、すべての候補節が競って適用条件を調べ、最も早く条件を満たした候補節が選択され、そのボディ部にある述語の計算に引き継がれる。言い替えるとヘッドで表される述語の計算は、ボディ部で表される述語群の計算で書き換えられる。そして書き換えられた述語群は同様にまたその述語定義中のボディ部の述語群で書き換えられていくのである。

KL1 の実行主体はゴールと呼ばれる。ボディ部の要素 $Body_i$ はすべてゴールであり、ボディ・ゴールと呼ばれることもある。

各ゴールは図 2 に示す実行イメージに従って実行される。

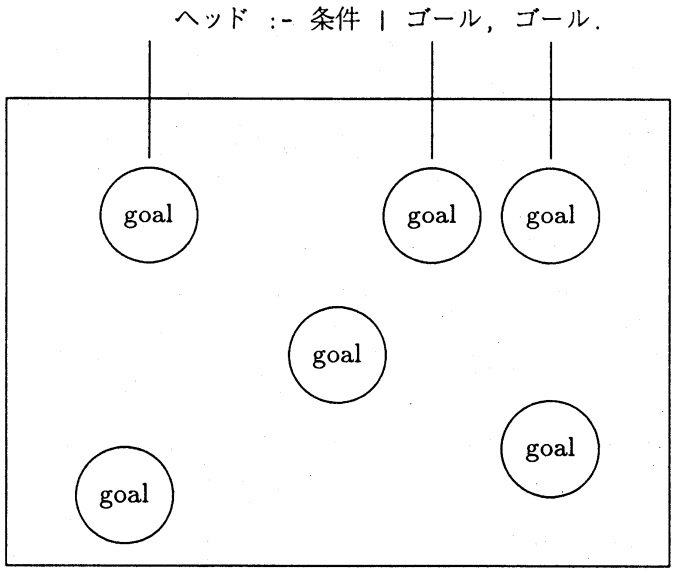

図 2　reduction プールと実行イメージ

プログラムの実行は、まず最初に初期ゴールを与え reduction プールに入れるところから始まる。まず reduction プールから適当にゴールを選び、図 2 の通り述語定義のヘッドおよび条件からなるガード部の条件が確認の後、ボディ部のゴール群に書き換えられる。する。書き換えられて生成されたボディゴールはすべて reduction プールに入れられ、同様に書き換えが繰り返される。書き換えが試みられるゴールの順番は規定されておらず、どのように行なわれても構わない。並列に実行されても構わない。ここに論理的並列性が存在する。基本的には reduction プールが空になればすべての書き換えが終了したことになる。述語定義を構成する候補節すべてのガード部の条件を満たさないことがわかった場合、そのプログラムの状態は failure となり、異常終了する。

各ヘッドや条件を構成する述語、ゴールを構成する述語は引数をもつことができる。これらの引数

間で直接もしくは間接的に共有する未定義データをもつことを述語定義で記述できる。各ゴールをスレッドと見立てれば、スレッド間で引数部で直接もしくは間接的に指される論理的な共有メモリをもつことが記述でき、この共有メモリ上にあるデータをアクセスし合うことで計算が行なわれるのである。この共有するメモリを論理変数と呼ぶ。KL1 では論理変数に対してデータの単一代入性が保証されている。ゴールの書き換えを通して、ヘッドからボディへ論理変数が伝達される。例として 2 つのリストの要素を 1 つのリストにまとめる append プログラムを図 3 に挙げる。大文字から始まる記号が論理変数を表し、名前は 1 つの節中でのみ有効であり、同じ論理変数を示す意味しかない。

```
append(AX, Y, AZ) :- AX = [A|X]  | AZ = [A|Z], append(X,Y,Z).
append(X,  Y,  Z) :- X = []      | Y = Z.
```

図 3 append プログラム

ガード部の '=' は passive unification と呼ばれ、この節によって具体化 (代入と言った方がイメージし易いかもしれない) できず、外部からの具体化により同じ構造のデータであることを確認することを意味する。もし、異なることが確認されたらこの候補節は選択されない。また論理変数であった場合は、まだ同じ構造であるかどうかが判断できないので、この候補節の書き換えは一時的に保留される。すべての候補節が異なった場合は、このゴールは失敗したと言う。すべての候補節の書き換えが保留された場合は、このゴール は suspend したと言う。またボディ部の '=' は active unification と呼ばれ、左辺と右辺を同一のデータとする。論理変数同士であれば以後同一の論理変数として扱われる。

例えば append([1,2],[3,4],A) を実行すると次の通りの結果が得られる。

```
append([1,2],[3,4],A)   →  A = [1|A1],   append([2],[3,4],A1)
append([2],[3,4],A1)    →  A1 = [2|A2],  append([],[3,4],A2)
append([],[3,4],A2)     →  A2 = [3,4]
```

A = [1,2,3,4] が得られる。append([1,2],Y,A) であれば A = [1,2 |Y] と未定義な部分を含む構造が結果となったり、append(X,[3,4],Z) であればこのゴールは suspend してしまう。

並列言語として考える。例えばappend([1,2],[3,4],Z), append(Z,[5,6],U) を実行する。この 2 つの append ゴールが別の worker で動作するとした時、Z は worker 間の通信路を意味する。1 つ目の append ゴールは Z に対して active unification を行なう。2 つ目の append ゴールは Z に対し passive unification つまり Z が外部から具体化されるのを待つことになる。つまり send と receive の同期関係が暗黙のうちに実現されているのである。1 つ目の append ゴールがリスト [1,2,3,4] を送信し、2 つ目の append ゴールが受信するのである。このように worker 間の同期通信がいとも簡単に記述できる。

■非決定的計算

並列論理型言語の特徴のひとつとして、非決定的計算の記述性の良さが挙げられる。図 4 のプログラムを見てみよう。

```
merge(AX, Y, AZ) :- AX = [A|X]  | AZ = [A|Z], merge(X,Y,Z).
merge(X,  Y,  Z) :-  X = []     | Y = Z.
merge(X, AY, AZ) :- AY = [A|Y]  | AZ = [A|Z], merge(X,Y,Z).
merge(X,  Y,  Z) :-  Y = []     | X = Z.
```

図 4　merge プログラム

例えば merge([1,2],[3,4],A) を実行するとどうなるであろう。例えば

merge([1,2],[3,4],A) → A = [1|A1] merge([2],[3,4],A1)

となるかもしれないし、また

merge([1,2],[3,4],A) → A = [3|A1] merge([1,2],[4],A1)

となるかもしれない。結果も A = [1,2,3,4] となるかもしれないし、また A = [3,1,2,4] となるかもしれない。何故かと言えば、この状況で選択できる候補節が複数あり、どの節が選ばれるかを規定してないからである[3]。図 4 では第 1 節および第 3 節が選択できる候補である。早いもの勝ちと簡単に考えても良い。特に並列マシンを想定すると第 1 引数と第 2 引数を具体化するどちらのゴールが先に実行されるかわからない場合もあるし、また通信の遅れも考慮するとどちらの節が選択されるかわからない。つまりこの述語定義には非決定性が記述されている。このように非決定性が簡単に記述できることも KL1 言語の特徴である。

この機能を使った面白いプログラムを図 5 に挙げる。

```
copy(AX, Stop, AZ) :- AX = [A|X]  | AZ = [A|Z], copy(X, Stop, Z).
copy(X,  Stop,  Z) :- X = []      | Z = [].
copy(X,  Stop,  Z) :- Stop = stop | Z = [].
```

図 5　copy プログラム

[3]実際には優先度をつけることが可能である。

`Stop = stop, copy([1,2], Stop ,Z)` を実行した場合、結果の Z は `[]`、`[1]` もしくは `[1,2]` のどれかになる。`Stop = stop` がいつ実行され、その結果がいつ copy に届くかによって計算結果が異なるのである。このように計算を途中で止めさせる非決定的なプログラムは、並列探索問題等で効果を発揮する。前節で述べた非決定的な数式計算も突き詰めればこのような KL1 言語では簡単に記述できることがお分かりになったと思う。

16.4 ポータブル KL1 処理系 KLIC

KL1 言語の実装のひとつに KLIC がある。KLIC は KL1 プログラムを C 言語に変換するコンパイラと実行時のサポートライブラリからなる。KLIC では KL1 言語で記述されたプログラムを C 言語に変換し、それを既存の C コンパイラやリンカを通じて通常の実行オブジェクトとして実行する。Lisp や Prolog に見られる独自の環境を構築し、その中でプログラムを操作するものはない。KLIC は逐次性能が高くかつ並列分散環境も含めてポータビリティが高いのが特徴である。

さて 16.2 節で挙げたような非決定的アルゴリズムを KLIC で実現することを考える。このような場合、代数計算ができる数式処理系は山ほどあるのにも関わらず、KL1 でまた書き直すのは面倒なものである。逆に普通の数式処理系でこのアルゴリズムを実現するのは非常に困難なものである。

実は KLIC は他言語プログラムをオブジェクト指向的な枠組で取り込むジェネリックオブジェクトと呼ばれる機能をもつ。ジェネリックオブジェクトは KL1 に対して次の機能を追加できる。

- データの拡張

  KLIC は基本データとしては 3 種類程度のものしか持たない。そこで KL1 言語として必要な他のデータを拡張するためにジェネリックオブジェクトを利用している。この機能はカーネルやコンパイラと独立なものであり、他言語[4]ユーザ定義が可能である。PARI の各データはこの機能で定義する。各処理についてはそれぞれメソッドを定義し、KL1 側から呼び出すことができる。

- ゴール (論理変数) の拡張

  KLIC で定義される述語相当の処理を他言語によりユーザ定義可能である。正確には論理変数を特殊化し、unification 等によって起動されるプログラムを他言語で記述することができる。もちろんある程度の規範に従って記述する必要があるが、KL1 のプログラミングスタイルとしてリストの CAR 部で同期をとり、CDR 部を自分自身に再書き換えして生成するゴールに渡すことで状態を実現することができる。この状態を刻々と変化させることは、通常の言語の副作用にあたる。KLIC では KL1 の枠組の中で通常言語の副作用を処理することができる。

記号処理言語処理系におけるこのような拡張の問題点は、メモリ管理にある。多くの記号処理言語処理系は自動メモリ管理機構を備え、ユーザをメモリ管理からの繁雑さから解放している。KLIC も

[4]多くの場合 C 言語である。

同様に自動メモリ管理機構を備え、いわゆるガーベジコレクション (以下 GC と略す) を行なう。ところがユーザ定義のデータを GC する場合、データの移動が起こる可能性がある。データ中にポインタがある場合にはその保守をする必要があるところが問題となる。KLIC のジェネリックオブジェクトはこの処理をユーザに委ねる、すなわち GC メソッドをユーザに定義させ、自分自身のデータに関する移動は自分自身で記述できるようにしている[5]。並列環境においてもある worker のデータをパケットに変換して他の worker に送り、受けとった worker 側でそのパケットからデータに戻す処理に関して、同様にデータ依存コードとなってしまう。encode メソッドを記述することでこの問題を解決できるのである。

16.5 PARI の並列化と計算例

PARI は冒頭で述べた通り、数式処理ライブラリであり、別のプログラムとリンクして利用する。前節の説明で明らかな通り、PARI のデータを KLIC のジェネリックオブジェクトとして実現することで簡単に PARI を利用するプログラムが並列化できる。また GP のように環境的な処理を導入する際は、KL1 で記述しても構わないが、今回はジェネリックオブジェクトのゴールの拡張機能で実現した。

さて最後に 16.2 節のアルゴリズムを計算した結果を [Murao & Fujise 94] より引用し、台数効果を表 2 に挙げる。逐次アルゴリズムにとって最良の因数の検索ができたとしても、並列化による弊害をほぼ除去できていることが確認できると思う。また逐次アルゴリズムにとって最悪の検索になった場合は、linear に近い台数効果が得られていることがおわかりになると思う。

表 2　100 次の多項式の因数分解 (2)

worker 数	1	2	4	8	12	16
逐次最良時の台数効果	1.0	1.0	1.0	0.9	1.0	1.0
逐次最悪時の台数効果	1.0	2.0	4.0	7.8	11.9	15.8

16.6 おわりに

並列代数計算アルゴリズムは、まだまだ未開拓な研究分野である。その理由のひとつとして実験環境が整ってないことが挙げられると思う。KLIC が普及することでこの分野の研究が活発になることを期待する。

---

[5] この考えは AGENTS[Janson 94] に基づく。

参考文献

[Cohen 93]Cohen, H.: A Course in Computational Algebaric Number Theory, Springer-Verlag, 1993.

[Fujise 94]Fujise, T. et al.: "KLIC: Portable Implementation of KL1", *Proc. of International Symposium on Fifth Generation Computer Systems 1994*, pp.66-79 (1994).

[Janson 94]Janson, S. et al.: AGENTS users manual, SICS technical report, SICS, 1994.

[Murao & Fujise 94]Murao, H. and Fujise, T.: Modular Algorithm for Sparse Multivariate Polynomial Interpolation and its Parallel Implementation, *Proc. of the Internatinal Symposium on Parallel Symbolic Computation 1994 (Hong,H. Ed.)*, pp.304-315 (1994).

[Ueda & Chikayama 90]Ueda, K. and Chikayama, T.: Design of the Kernel Language for thr Parallel Inference Machine, *The Computer Journal*, Vol.33 No.6, pp.494-500 (1990).